1、栈定义

栈是一种遵从后进先出（LIFO）原则的有序集合。

创建栈

```
//使用类来创建栈
function Stack(){
//各种属性和方法的声明
}
```

栈常见方法

```
push(elements) //添加一个或多个元素
pop() //移除栈顶元素
peek() //返回栈顶元素
isEmpty() //栈空返回true
clear() //移除栈里的所有元素
size() //返回栈里元素个数
```

栈常见方法实现

```
function Stack(){
    //各种属性和方法的声明
    let items=[]//使用数组来保存栈里的元素
    this.push=function(elements){
        items.push(elements)
    }
    this.pop=function(){
        return items.pop()
    }
    this.peek=function(){
        return items[items.length-1] // 数组中最后一个元素
    }
    this.isEmpty=function(){
        return items.length===0
```

```
    }
    this.size=function(){
        return items.length
    }
    this.clear=function(){
        items=[]
    }
    this.print=function(){
        console.log(items.toString())
    }
}
```

栈的运用常景

例子：将十进制数字10转换成二进制数字

```
function Tento(num){
    let stack = new Stack()
    console.log(stack)
    let rem//余数
    let str = ''
    do{
        rem = Math.floor(num % 2)
        console.log('余数: '+rem)
        stack.push(rem)
        num = Math.floor(num / 2)//向下取整
        console.log(num)
    }while(num > 0)
    //当num大于0时一直进行do操作
    while(stack.isEmpty() === false){
        //判断栈是否为空，不为空一直输出
        str += stack.pop().toString()
    }
    return str
}
console.log(Tento(2))
```

原文链接：https://blog.csdn.net/weixin_40119412/article/details/108294529

2、队列定义

队列是遵循先进先出（FIFO)原则的一组有序的项。

创建队列

```
function Queue(){
//队列的属性和方法声明
}
```

队列常见方法

```
enqueue(elements) //向队尾添加一个或多个元素
dequeue() //移除队头元素，并返回被移除的元素
front() //返回队列中第一个元素
isEmptzy() //如果队列不包含任何元素，返回true
size() //返回队列保护的元素个数
```

队列常见方法实现

```
function Queue(){
    //队列的属性和方法声明
    //使用数组来存储队列元素
    let items=[]
    this.enqueue=function(elements){
        items.push(elements)
    }
    this.dequeue=function(){
        return items.shift()
    }
    this.front=function(){
        return items.length===0
    }
    this.isEmpty=function(){
```

```
        return items.length===0
    }
    this.size=function(){
        return items.length
    }
    this.print=function(){
        console.log(items.toString())
    }
}
```

优先队列

```
function PriorityQueue(){
    var items=[]
    //创建元素 带上优先级
    function QueueElement(element,priority){
        this.element=element;
        this.priority=priority;
    }
    this.enqueue=function(element,priority){
    var queueElement=new QueueElement(element,priority);
    //如果队列为空，则直接添加
    if(this.isEmpty()){
        items.push(queueElement)
    }else{
    //判断是否排队
        var added=false;//不排队
        for(var i=0;i<items.length;i++){
    //比较优先级
        if(queueElement.priority<items[i].priority){
            items.splice(i,0,queueElement);
                added=true;
                break;
            }
            排队，直接在后面添加
                if(!added){
                    items.push(queueElement);
                }
```

```
            }
        }
    }
    //其他方法与普通队列一样
}
```

循环队列

例子：击鼓传花

```
function hotPotato(nameList,num){
    //nameList表示总数,num表示次数
    var queue=new Queue();
    for(var i=0;i<nameList.length;i++){
        queue.enqueue(nameList[i]);
    }
    var eliminated='';//被淘汰的
    while(queue.size()>1){
        for(var i=0;i<num;i++){
            queue.enqueue(queue.dequeue());
    }
    eliminated=queue.dequeue();
        console.log(eliminated+'failed);
    }
    return queue.dequeue()//返回被删除的元素
```

3、集合定义

集合是由一组无序且唯一的项组成的。

创建集合

```
function Set(){
 var items={}
}
```

集合常见方法

```
add(value) //向集合中添加一个新的项
remove(value) //从集合中移除一个项
has(value) //如果该值存在集合中，返回true
clear() //清空集合
size() //返回集合所包含元素的数量
values() //返回一个包含集合中所有值的数组
```

常见方法实现

has方法

```
//第1种
this.has=function(value){
 return value in items;
}

 //第2种
this.has=function(value){
return items.hasOwnProperty(value)
```

往集合中添加新的项，首先要判断集合中是否已经存在该项

```
this.add=function(value){
        //不存在，则添加
 if(!this.has(value)){
  items[value]=value;
  return true;
 }
 return false;
}
```

从集合中删除某个项，首先要判断集合中是否已经存在该项

```
this.remove=function(value){
 if(this.has(value)){
  delete items[value];
```

```
    return true;
   }
   return false;
  }
```

清除集合中所有 的项

```
this.clear=function(){
 items={}
}
```

集合的属性

```
this.size=function(){
 return Object.keys(items).length
}

 this.size=function(){
 var count=0;
 for(var prop in items){
  if(items.hasOwnProperty(prop))
  ++count;
 }
 return count;
}
```

返回集合所有元素

```
//Object.keys()返回包含对象所有属性的数组
 this.values=function(){
  return Object.keys(items)
 }

  this.value=function(){
  var keys=[];
  for(var key in items){
   keys.push(key);
  }
  return keys;
```

```
}
```

集合的操作

并集：对于给定的两个集合，返回一个包含两个集合中所有元素的新集合

实现

```
this.union=function(otherSet){
 let unionSet=new Set()
 let values=this.values()//值
 for(let i=0;i<values.length;i++){
  unionSet.add(values[i])
 }
 values=otherSet.values()
 for(let i=0;i<values.length;i++){
  unionSet.add(values[i])
 }
 return unionSet
}
```

代码测试

```
var setA=new Set();
setA.add(3)
setA.add(5)
setA.add(15)
var setB=new Set();
setB.add(3)
setB.add(9)
setB.add(15)
let unSet=setA.union(setB)
console.log(unSet.values())//(4) ["3", "5", "9", "15"]
```

交集：对于给定的两个集合，返回一个包含两个集合中共有元素的新集合

实现

```
this.intersection=function(otherSet){
 let intersection=new Set()
 let values=this.values()
```

```
  for(let i=0;i<values.length;i++){
   if(otherSet.has(values[i])){
    intersection.add(values[i])
   }
  }
  return intersection
 }
```

代码测试

```
var setA=new Set();
setA.add(3)
setA.add(5)
setA.add(15)
var setB=new Set();
setB.add(3)
setB.add(9)
setB.add(15)
let interSet=setA.intersection(setB)
console.log(interSet.values())
```

差集：对于给定的两个集合，返回一个包含所有存在于第一个集合且不存在于第二个集合的元素的新集合

实现

```
 this.difference=function(otherSet){
  let difference=new Set()
  let values=this.values()
  for(let i=0;i<values.length;i++){
   if(otherSet.has(values[i])===false){
    difference.add(values[i])
   }
  }
  return difference
 }
```

代码测试

```
var setA=new Set();
```

```
setA.add(3)
setA.add(5)
setA.add(15)
var setB=new Set();
setB.add(3)
setB.add(9)
setB.add(15)
let difference=setA.difference(setB)
console.log(difference.values())
```

子集：验证一个给定集合是否是另一个集合的子集

实现

```
 this.subset=function(otherSet){
      if(this.size()>otherSet.size()){
       return false
      }else{
           let values=this.values()
           for(let i=0;i<values.length;i++){
                if(otherSet.has(values[i])===false){
                     return false
                }
           }
           return true
          }
 }
```

代码测试

```
var setA=new Set();
setA.add(3)
setA.add(5)
setA.add(15)
var setB=new Set();
setB.add(3)
setB.add(9)
setB.add(15)
console.log(setA.subset(setB))//false
```